Folded FIFOs

Rajit Manohar

Computer Science Department
California Institute of Technology

1. REPORT DATE **10 JUL 1995**	2. REPORT TYPE	3. DATES COVERED **10-07-1995 to 10-07-1995**
4. TITLE AND SUBTITLE **Folded FIFOs**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **11**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Folded FIFOs

Rajit Manohar*
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125.

July 10, 1995

Abstract: *We present two distributed implementations of first-in first-out message buffers. The solutions presented reduce the delay between insert and delete operations on the buffer when the buffer is empty. The designs are then modified so as to offer bounded-response-time. The solutions presented use a CSP-like notation and are suitable for transformation into a VLSI circuit.*
Keywords: *distributed systems, systolic designs, bounded-response-time, maximum storage utilization, buffers, VLSI.*

1. Introduction.

A folded FIFO is an implementation of a distributed first-in first-out message buffer. Two operations are permitted: a *push* operation inserts data into the FIFO; a *pop* operation removes a data item from the FIFO. Ordinarily, a distributed FIFO consists of a series of one-place buffers. If the FIFO remains relatively empty, then the delay between a *push* and a *pop* operation is proportional to the size of the buffer. In a folded FIFO, we try to short-circuit the path taken by the data so that this delay is reduced. Fig. 1 shows the way in which data can move in a folded FIFO.
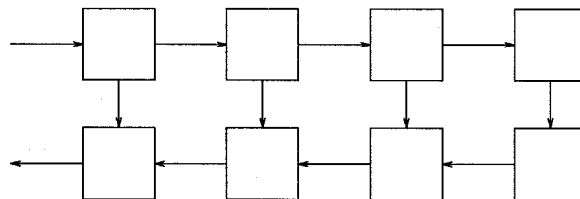

Fig. 1: Folded FIFO data flow.

In this paper, we describe two implementations of a folded FIFO using CSP-like specifications (cf. [1]) with probes (cf. [3]). We provide informal arguments justifying the correctness of the designs. The designs presented are systolic and distributed, which makes them suitable candidates for transformation into a VLSI circuit (cf. [4]).

2. Bounded-Response-Time FIFOs.

A design for a FIFO has a bounded-response-time if the time between two successive permissible operations is independent of the capacity of the FIFO. A detailed discussion of this phenomenon can be found in [2], where it is also shown that there does not exist a distributed FIFO implementation that has both bounded-response-time and maximum storage utilization. This fact is based on the following three assumptions:

1. Movement of data in one direction corresponds to movement of a vacancy in the opposite direction.

2. A vacancy can cover only a bounded path in bounded time.

3. The number of paths not exceeding a given length between the external *push* and *pop* port is bounded.

It is unreasonable to assume that the second assumption is violated in any implementation of a FIFO. The third assumption cannot be violated if every cell has a fixed, finite number of neighbors (as in a VLSI implementation). However, the first assumption can be violated by allowing swap operations. Using swaps, there are implementations of a folded FIFO with constant-response-time. Intuitively, a swap operation introduces temporary storage which violates the maximum storage utilization restriction.

We can either try to maintain a low-energy configuration (described below) in which we do not move a data item unless we are forced to do so, or we can aim for an implementation that has bounded-response-time. However, we know that there is no implementation of a FIFO that has bounded-response-time and maximum storage utilization. Therefore we explore the possibility of constructing a folded FIFO with a low-energy configuration.

For low energy, we must avoid moving data unnecessarily (cf. [5], [6]). This means that we must keep the folded FIFO in a compact configuration, i.e. we must not have any "holes" in the FIFO. Fig. 2 shows a configuration that should not be permitted, where the shaded cells are those that contain valid data.
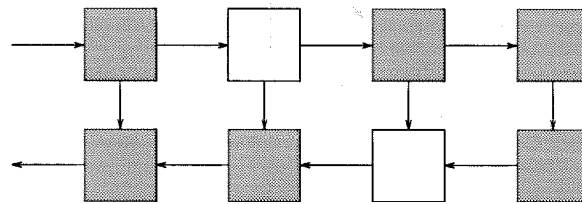

Fig. 2: Folded FIFO with "holes."

Suppose a series of *pop* operations were performed from the configuration shown in Fig. 2. Then, the data items in the cells after the holes would have moved through more cells than necessary, resulting in a waste of energy.

3. Solution 1.

If we assume that *push* and *pop* are mutually exclusive, then we can combine the two processes that are above one another (cf. Fig. 1) into a single process. We obtain the process structure shown in Fig. 3.
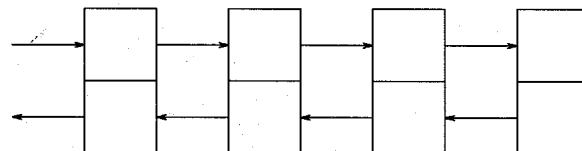

Fig. 3: Folded FIFO—single process.

Notice that we have potentially sacrificed 50% throughput by making this assumption since we can no longer have concurrent *push* and *pop* operations.

For minimum energy, we will try to maintain the following invariant: If a process does not hold two

data items, no process to its right holds any data items. Keeping this invariant in mind, we now describe the possible states for a folded FIFO cell (cf. Fig. 4).
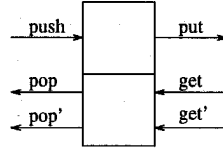

Fig. 4: Folded FIFO cell—single process

The FIFO cell can be in one of four different states:

- $\langle empty \rangle$. The cell has no data.
- $\langle half \rangle$. The cell has one data item (stored in x).
- $\langle full \wedge last \rangle$. The cell has two data items (x has been inserted after y) and every cell to its right is empty.
- $\langle full \wedge \neg last \rangle$. The cell has two data items (x has been inserted after y) and the cell to its right is not empty.

Given these states, CSP for the cell is straightforward and is given below. For clarity, we write the specification in terms of state transitions. Each action is terminated with the next state of the cell. Note that in fact the specification indicates a single non-terminating process.

The only interesting case is when a *pop* operation is performed and the cell is in state $\langle full \wedge \neg last \rangle$. To determine the next state, it is necessary to know if the next cell (to the right) is empty. This information is sent with the data along channel pop'.

$$\langle empty \rangle \equiv push?x; \quad \langle half \rangle$$

$$\langle half \rangle \equiv [\ \overline{push} \longrightarrow y := x; \quad push?x; \quad \langle full \wedge last \rangle$$
$$\quad [\ \overline{pop'} \longrightarrow pop'!true; pop!x; \quad \langle empty \rangle$$
$$\quad]$$

$$\langle full \wedge last \rangle \equiv [\ \overline{push} \longrightarrow put!x; \quad push?x; \quad \langle full \wedge \neg last \rangle$$
$$\quad [\ \overline{pop'} \longrightarrow pop'!false; pop!y; \quad \langle half \rangle$$
$$\quad]$$

$$\langle full \wedge \neg last \rangle \equiv [\ \overline{push} \longrightarrow put!x; \quad push?x; \quad \langle full \wedge \neg last \rangle$$
$$\quad [\ \overline{pop'} \longrightarrow pop'!false; pop!y; \quad get'?b; get?y; \quad [b \longrightarrow \langle full \wedge last \rangle \,[\,\neg b \longrightarrow \langle full \wedge \neg last \rangle\,]$$
$$\quad]$$

Remark. Since the operations on *pop* and *pop'* (and similarly *get* and *get'*) are always joint, we could combine the two channels into a single channel that transmits a pair. (*End of Remark.*)

The additional copying involved in the assignment $y := x$ can be avoided by adding additional states in which only y contains data, or y has been inserted before x in the FIFO. The last cell can be obtained by assuming that *put* and *get* operations will deadlock when attempted in the last cell. The *pop'* channel can be dropped from the first process by replacing probes on *pop'* with probes on *pop*.

Although this solution does have low energy, notice that the FIFO will deadlock if a *push* is performed when the FIFO is full (or if a *pop* is done when the FIFO is empty). This is a direct result of the assumption that *push* and *pop* operations are mutually exclusive, which allowed us to combine two processes into a single cell. We now present another solution that does not suffer from this problem.

4. Solution 2.

The cell for the second solution contains the channels shown in Fig. 5.
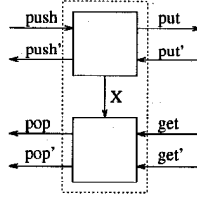

Fig. 5: Folded FIFO cell—concurrent *push*, *pop*.

Two additional channels are used to propagate information backward in the FIFO. In addition, the data channel X is introduced for communication within a cell. The dotted line around the processes indicates that the cell begins as one process, and then may fork into two processes which can then recombine into a single process.

The primed channels are used to indicate that a process is ready for communication, and the value passed along the channel indicates how the communication action is to continue. A *true* value indicates that the initiated data action should continue. A *false* value indicates that the data should take an alternative route.

Once again we maintain the invariant that if the cell is either in the ⟨*empty*⟩ or ⟨*half*⟩ state, then every cell to its right is empty. The sequential states of the cell are as follows:

- ⟨*empty*⟩. The cell has no data.
- ⟨*half*⟩. The cell has one data item stored in x.
- ⟨*split*⟩. The cell has two data items. This state is split into two processes:
    - ⟨*top*⟩ for the process representing the top cell, and
    - ⟨*bottom*⟩ representing the cell on the bottom.

We use the single vertical bar "|" to denote non-deterministic choice (i.e. requiring arbitration). Finally, notice that the CSP for the ⟨*half*⟩ state contains a fork. The join for this fork is in the ⟨*top*⟩ and ⟨*bottom*⟩ states—they return to the ⟨*half*⟩ state concurrently, since this state change is preceeded by a communication on channel X.

Given the meaning of the primed channels, the CSP for the various states is almost straightforward. In the ⟨*empty*⟩ state, a *push'* can continue, which corresponds to sending a value of *true* along *push'* and then accepting a data item. A *pop* operation in this state cannot succeed, and therefore returns the value *false*.

$$\langle empty \rangle \equiv [\ \overline{push'} \longrightarrow push'!true; push?x;\ \langle half \rangle$$
$$|\ \overline{pop'} \longrightarrow pop'!false;\ push'!false;\ \langle empty \rangle$$
$$]$$

In state $\langle half \rangle$, one can accept both a *push* and a *pop* operation, and the CSP for this state is given below. Notice that on accepting a *push*, we enter the split state allowing concurrent *push* and *pop* operations.

$$\langle half \rangle \equiv [\ \overline{push'} \longrightarrow\ push'!true; push?y;\ \langle top \rangle\ \|\ \langle bottom \rangle;\ \langle half \rangle$$
$$|\ \overline{pop'} \longrightarrow pop'!true; pop!x;\ \langle empty \rangle$$
$$]$$

The $\langle bottom \rangle$ state is simple. Since it only has to respond to *pop* requests and it contains data, it begins by accepting a *pop* operation. Finally, it requests data from the cell to its right. If the *pop* succeeds, then it remains in the $\langle bottom \rangle$ state. Otherwise, it returns to the $\langle half \rangle$ state by accepting data from the cell on top.

$$\langle bottom \rangle \equiv pop'!true; pop!x;\ get'?b;\ [\ b \longrightarrow get?x;\ \langle bottom \rangle\ [\!]\ \neg b \longrightarrow X?x;\ \langle skip \rangle\ ]$$

The final state is the most complex. In this state, a cell has to accept two requests; an external *push* request, and an X request.

$$\langle top \rangle \equiv [\ \overline{push'} \longrightarrow put'?b;\ [\ b \longrightarrow push'!true;\ put!y;\ push?y;\ \langle top \rangle$$
$$[\!]\ \neg b \longrightarrow X!y;\ \langle skip \rangle$$
$$]$$
$$|\ \overline{X} \longrightarrow put'?b;\ X!y;\ \langle skip \rangle$$
$$]$$

Notice that this solution has two arbiters: one arbitrating between $\overline{push'}$ and $\overline{pop'}$ in $\langle half \rangle$, and one arbitrating between $\overline{push'}$ and $\overline{X}$ in $\langle top \rangle$. The first arbiter between $push'$ and pop' cannot be avoided since we are allowing concurrent *push* and *pop* operations. However, we make the following observation: in the CSP for $\langle top \rangle$, after probing X we *know* that the value returned in b will be *false*. Observing the similarity between the two guarded commands, we can rewrite $\langle top \rangle$, removing the arbiter!

$$\langle top \rangle \equiv [\overline{push'} \vee \overline{X}];\ put'?b;\ [\ b \longrightarrow push'!true;\ put!y;\ push?y;\ \langle top \rangle$$
$$[\!]\ \neg b \longrightarrow X!y;\ \langle skip \rangle$$
$$]$$

Finally, the last cell can be obtained by assuming that $*[push'!false]\ \|\ *[pop'!false]$ is the process to its right. For the leftmost process, the *push'* and *pop'* channels can be removed by replacing the probes with probes on *push* and *pop*. Finally, the *pop'* selection in the leftmost $\langle empty \rangle$ process should be removed.

5. Bounded-Response-Time Solutions.

We can make either of these solutions bounded-response-time by introducing temporary storage, if we assume that the environment does not try to insert items into a full buffer. We do so by introducing a variable t. However, note that this variable is written and read exactly once. This allows us to implement variable t efficiently in a VLSI circuit.

In the first solution, a *pop* is constant-response-time. A *push* is not constant-response-time only if the FIFO cell is full. We modify the full state as follows:

$$\langle \mathit{full} \wedge \mathit{last} \rangle \equiv [\; \overline{\mathit{push}} \longrightarrow t := x; \;\; \mathit{put}!t \bullet \mathit{push}?x; \;\; \langle \mathit{full} \wedge \neg \mathit{last} \rangle$$
$$[]\; \overline{\mathit{pop'}} \longrightarrow \mathit{pop'}!\mathit{false}; \mathit{pop}!y; \;\; \langle \mathit{half} \rangle$$
$$]$$

$$\langle \mathit{full} \wedge \neg \mathit{last} \rangle \equiv [\; \overline{\mathit{push}} \longrightarrow t := x; \;\; \mathit{put}!t \bullet \mathit{push}?x; \;\; \langle \mathit{full} \wedge \neg \mathit{last} \rangle$$
$$[]\; \overline{\mathit{pop'}} \longrightarrow \mathit{pop'}!\mathit{false}; \mathit{pop}!y; \;\; \mathit{get'}?b; \mathit{get}?y; \;\; [\; b \longrightarrow \langle \mathit{full} \wedge \mathit{last} \rangle \;\; [] \; \neg b \longrightarrow \langle \mathit{full} \wedge \neg \mathit{last} \rangle \;\;]$$
$$]$$

In the second solution, a *pop* is constant-response-time. Using a strategy similar to the one outlined above is not enough since we have to make both *push* and *push'* constant-response-time. To make *push'* constant-response-time in process $\langle \mathit{top} \rangle$, we must complete the *push'* operation by sending it a *true* value. Unfortunately, this prevents us from removing the arbiter between $\overline{X}$ and $\overline{\mathit{push'}}$. As a start, suppose we complete the *push'* operation early.

$$\langle \mathit{top} \rangle \equiv [\; \overline{\mathit{push'}} \longrightarrow \mathit{push'}!\mathit{true}; \;\; \mathit{put'}?b; \;\; [\; b \longrightarrow t := y; \;\; \mathit{put}!t \bullet \mathit{push}?y; \;\; \langle \mathit{top} \rangle$$
$$[]\; \neg b \longrightarrow X!y; \;\; \langle ?? \rangle$$
$$]$$
$$|\; \overline{X} \longrightarrow \mathit{put'}?b; \;\; X!y; \;\; \langle \mathit{skip} \rangle$$
$$]$$

Originally, we entered the $\langle \mathit{half} \rangle$ state after $X!y$ and the *push* operation was completed there. Since we have completed *push'*, we must also complete *push*. As a result, we need to distinguish between the two different X actions. To do so, additional information is sent along X. Finally, $\langle \mathit{bottom} \rangle$ is modified to reflect these changes. The final solution is:

$$\langle \mathit{top} \rangle \equiv [\; \overline{\mathit{push'}} \longrightarrow \mathit{push'}!\mathit{true}; \;\; \mathit{put'}?b; \;\; [\; b \longrightarrow t := y; \;\; \mathit{put}!t \bullet \mathit{push}?y; \;\; \langle \mathit{top} \rangle$$
$$[]\; \neg b \longrightarrow X!(\mathit{true}, y); \;\; \mathit{push}?y; \;\; \langle \mathit{top} \rangle$$
$$]$$
$$|\; \overline{X} \longrightarrow \mathit{put'}?b; \;\; X!(\mathit{false}, y); \;\; \langle \mathit{skip} \rangle$$
$$]$$

$$\langle \mathit{bottom} \rangle \equiv \mathit{pop'}!\mathit{true}; \mathit{pop}!x; \;\; \mathit{get'}?b; \;\; [\; b \longrightarrow \mathit{get}?x; \;\; \langle \mathit{bottom} \rangle$$
$$[]\; \neg b \longrightarrow X?(c, x); \;\; [\; c \longrightarrow \langle \mathit{bottom} \rangle \;\; [] \; \neg c \longrightarrow \langle \mathit{skip} \rangle \;\;]$$
$$]$$

6. Correctness Proof.

For the arguments outlined below, we will use the following terminology. An *invariant* will be used to refer to a *loop invariant*. In other words, the "invariants" will hold only at the beginning of any state. We will assume that s is a ghost variable that indicates the current state. $e.x$ is a boolean value that is true if and only if data register x is empty, and $f.x$ is defined to be $\neg e.x$. We assume that each data value has

associated with it a sequence number, and that $ord.x$ denotes the sequence number of x. In other words, data value x corresponds to the data that was inserted by the $ord.x^{\text{th}}$ *push* operation. Note that $ord.x$ is defined if and only if $f.x$ holds. For a cell c we use $c.v$ to denote a variable in cell c. rt is the cell immediately to the right of the current cell, and lt is the cell to the left. We will use "pre state" to refer to the stable state before any actions were executed, and "post state" to refer to the stable state after the actions mentioned were executed.

We have two proof obligations: (I) there is no deadlock amongst the processes; and (II) the kth *pop* operation receives the data that was *pushed* by the kth *push* operation. We now provide correctness arguments for the first two solutions that were presented.

SOLUTION 1.

The following invariants are used to justify the names of the states.

$$s = \langle empty \rangle \Rightarrow (e.x \wedge e.y \wedge rt.s = \langle empty \rangle) \tag{1}$$

$$s = \langle half \rangle \Rightarrow (f.x \wedge e.y \wedge rt.s = \langle empty \rangle) \tag{2}$$

$$s = \langle full \wedge last \rangle \Rightarrow (f.x \wedge f.y \wedge rt.s = \langle empty \rangle) \tag{3}$$

$$s = \langle full \wedge \neg last \rangle \Rightarrow (f.x \wedge f.y \wedge rt.s \neq \langle empty \rangle) \tag{4}$$

The following invariant indicates the FIFO nature when a cell is full.

$$s = \langle full \wedge last \rangle \Rightarrow ord.y + 1 = ord.x \tag{5}$$

The following two invariants indicate the FIFO nature when $s = \langle full \wedge \neg last \rangle$.

$$s = \langle full \wedge \neg last \rangle \wedge rt.s = \langle half \rangle \Rightarrow (ord.(rt.x) + 1 = ord.x \wedge ord.y + 1 = ord.(rt.x)) \tag{6}$$

$$s = \langle full \wedge \neg last \rangle \wedge f.(rt.y) \wedge f.(rt.x) \Rightarrow (ord.(rt.x) + 1 = ord.x \wedge ord.y + 1 = ord.(rt.y)) \tag{7}$$

> *Remark.* The rightmost cell can never enter the state $\langle full \wedge \neg last \rangle$ since communication on
> *put* deadlocks. (*End of Remark.*)

Initially, every cell is in state $\langle empty \rangle$, and $e.x \wedge e.y$ holds in each cell. Therefore all the invariants hold. Note that for a cell to change state, a communication action must be performed with the cell.

Notice that the value sent on channel pop' is *true* if and only if the next state is $\langle empty \rangle$. This fact will be necessary to establish the state invariants from state $\langle full \wedge \neg last \rangle$.

We consider the actions from every state and demonstrate that all the invariants hold. We will ignore invariants that hold trivially in the post state.

> $s = \langle empty \rangle$. Only a *push* operation can be performed. Now, $f.x$ will hold after $push?x$. Since (1) holds in the pre state, we conclude that (2) holds in the post state.
>
> $s = \langle half \rangle$.
>
>> *pop'*. Since sending data empties a register, and using (2) in the pre state, we can conclude that (1) holds in the post state, in which we enter state $\langle empty \rangle$.
>>
>> *push.* On completion, we enter state $\langle full \wedge last \rangle$. Clearly the previous cell was in state $\langle full \wedge \neg last \rangle$ before it began the actions leading to the *push* operation (from the state invariants). Since

the invariants (6), (4) held in the pre state for the previous cell and $e.y$ was true in the pre state, $ord.y + 1 = ord.x$ holds in the post state since x was obtained from x in the left cell. For the leftmost cell, $ord.y + 1 = ord.x$ holds because from the definition of ord. Also, $f.x \wedge f.y$ holds in the post state. Finally, (2) in the pre state implies that $rt.s = \langle empty \rangle$. Therefore we have (3) and (5) in the initial state of $\langle full \wedge last \rangle$.

$s = \langle full \wedge last \rangle$.

pop'. In the post state, $e.y$ holds. Also, in the pre state, $ord.y + 1 = ord.x$ from (5). Examining the pre state of the cell on the left, we note that in its post state, (6) will hold since y is copied into $lt.y$. Finally, (3) in the pre state implies (2) in the post state.

$push$. In the post state, $rt.(f.x)$ holds since in the pre state $rt.s = \langle empty \rangle$ by (3). The invariant (7) holds in the post state because (7) was true in lt in the pre state for lt. Finally, (4) holds because rt will change to a non-empty state.

$s = \langle full \wedge \neg last \rangle$.

pop'. Sending y to the left maintains (7) for the left cell because (7) holds in in the pre state. Finally, the get operation maintains (7) in the post state because of (7) in the pre state. Finally, $rt.s = \langle empty \rangle$ holds if and only if we change to $\langle full \wedge last \rangle$ because of the observation made about the pop' channel.

$push$. Similar to the reasoning for $\langle full \wedge last \rangle$.

Note that $push$, pop operations can never deadlock since they are probed. Also, each channel is only probed at one end. A cell can never be suspended at a get operation (from the state invariants). Suppose the cell is suspended at a put operation. A put suspends forever if and only if the cell to its right is also suspended. Therefore the cell to the right must be suspended on a put as well. Inducting to the right, we note that every cell must have $f.x \wedge f.y$. Using the state invariants, we also note that every cell to the left must also have $f.x \wedge f.y$ to be true. Therefore, the FIFO is full. Therefore, suspension implies the FIFO is full. An external pop suspends if the FIFO is empty since the $\langle empty \rangle$ state only accepts a $push$ operation.

Finally, the invariants (5), (6), and (7) together imply that the kth pop operation receives data entered by the kth $push$ operation.

SOLUTION 2.

We will consider $\langle split \rangle$ to be the state in which we have two processes. The following invariants are used to justify the names of the states.

$$s = \langle empty \rangle \Rightarrow (e.x \wedge e.y \wedge rt.s = \langle empty \rangle) \tag{8}$$

$$s = \langle half \rangle \Rightarrow (f.x \wedge e.y \wedge rt.s = \langle empty \rangle) \tag{9}$$

$$s = \langle split \rangle \Rightarrow (f.x \wedge f.y) \tag{10}$$

Finally, we have the following three invariants:

$$s = \langle split \rangle \wedge rt.s = \langle empty \rangle \Rightarrow (ord.x + 1 = ord.y) \tag{11}$$

$$s = \langle split \rangle \wedge rt.s = \langle half \rangle \Rightarrow (ord.(rt.x) + 1 = ord.y \wedge ord.x + 1 = ord.(rt.x)) \tag{12}$$

$$s = \langle split \rangle \wedge rt.s = \langle split \rangle \Rightarrow (ord.(rt.y) + 1 = ord.y \wedge ord.x + 1 = ord.(rt.x)) \qquad (13)$$

Notice that a *false* value is sent along a primed channel if and only if the cell is in the $\langle empty \rangle$ state and a *pop'* was requested. Furthermore, note that if $\overline{X}$ is *true* in $\langle top \rangle$ *before* a *put'* operation is performed, then this means that a *put'* operation must succeed and return *false*. Also, if a *true* value is sent along a primed channel, then the data action along that channel must succeed. We will use these observations in the argument outlined below.

$s = \langle empty \rangle$.

> *push'*. Entering state $\langle half \rangle$, we must check that (12) and (9) are maintained. The latter holds trivially since no communication is performed with the *rt* cell, and (8) was true in the pre state. (12) holds because (11) was true in the pre state for *lt*.

> *pop'*. All invariants are maintained trivially.

$s = \langle half \rangle$.

> *push'*. We may violate (11), (10), or (13). (10) holds trivially. (11) holds for the cell because (12) was true for *lt*. Finally, (13) holds for *lt* because (12) holds in *lt*, and (13) was true in the cell to the left of *lt*.

> *pop'*. Clearly (8) holds in the post state from (9). From (12) in the pre state for *lt*, we can conclude that (11) holds in *lt* in the post state of *lt*.

$s = \langle split \rangle$.

> $\langle bottom \rangle$. Consider the execution sequence reaching $\langle bottom \rangle$. This satisfies all invariants in the post state for reasons similar to those in the previous solution. Consider the execution sequence reaching $\langle half \rangle$. Since *get'* returned *false*, it follows that $rt.s = \langle empty \rangle$. Therefore, (9) is maintained. In addition, this means that (11) held in the pre state, and so we can conclude that (12) holds for *lt* in the post state for *lt*.

> $\langle top \rangle$. Consider the execution sequence reaching $\langle top \rangle$. It satisfies all the invariants in the post state for reasons similar to those in the previous solution. Consider the execution sequence reaching $\langle half \rangle$. Using the arguments for $\langle bottom \rangle$, once again we conclude that the invariants are maintained.

Notice that *push* and *pop* actions can never suspend because of the observations made earlier. Similarly, X can never suspend. A *pop'* can never suspend. Therefore the only possible suspension is on a *push'* in state $\langle split \rangle$. Therefore the cell must be suspended on a *put'* (otherwise the *push'* can be completed eventually). Inducting, we see that all cells to the right must be suspended on a *put'*. Also, the invariants imply that all cells to the left must be in the $\langle split \rangle$ state. Therefore, the entire FIFO is full. Therefore we have absence of deadlock in the case that the FIFO is not full. When the FIFO is full, a *push'* can suspend. However, it will only suspend until a *pop'* operation is performed.

Finally, the FIFO property follows from the FIFO invariants, namely (11), (12), and (13).

7.   Conclusion.

We have presented two solutions to the folded FIFO problem with arguments justifying their correctness.

Both of them have a low energy configuration. Given the three restrictions from [2], it is known that there does not exist a FIFO implementation with constant-response-time. However, by introducing temporary storage, we have demonstrated that the low energy solutions can be converted into ones with bounded-response-time.

Appendix

The notation we use is based on Hoare's CSP [1]. A full description of the notation and its semantics can be found in [4]. What follows is a short and informal description of the notation we use.

- Assignment: $a := b$. This statement means "assign the value of b to a."
- Selection: $[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard. The notation $[G]$ is short-hand for $[G \rightarrow skip]$. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the *true* guards and executing the corresponding statement, repeating this until all guards evaluate to *false*. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is *true* iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$.

References

[1] C.A.R. Hoare. Communicating Sequential Processes. *CACM* 21(8):666-677, 1978

[2] J.L.W. Kessels and M. Rem. Designing systolic, distributed buffers with bounded response time. *Distributed Computing* 4(1) pp. 37–43, 1990.

[3] A.J. Martin. The Probe: An addition to Communication Primitives. *Information Processing Letters* 20:125–130, 1985.

[4] A.J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, 1(4), 1986.

[5] J.A. Tierno. An Energy-Complexity Model for VLSI Computations. PhD Thesis, Caltech. January 1995.

[6] J.A. Tierno and A.J. Martin. Low-energy asynchronous memory design. *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.